ANALYSIS ON AN IMPLEMENTATION OF THE GENS-DOMINGOS SUM-PRODUCT NETWORK STRUCTURAL LEARNING SCHEMA

Renato Lui Geh
Computer Science
Institute of Mathematics and Statistics
University of São Paulo
`renatolg@ime.usp.br`

Abstract. Sum-Product Networks (SPNs) are a class of deep probabilistic graphical models. Inference in them is linear in the number of edges of the graph. Furthermore, exact inference is achieved, in a valid SPN, by running through its edges twice at most, making exact inference linear. The Gens-Domingos SPN Schema is an algorithm for structural learning on such models. In this paper we present an implementation of such schema, analyzing its complexity, discoursing implementational and theoretical details, and finally presenting results and experiments achieved with this implementation.

Keywords `cluster analysis; data mining; probabilistic graphical models; tractable models; machine learning; deep learning`

1. Introduction

A Sum-Product Network (SPN) is a probabilistic graphical model that represents a tractable distribution of probability. If an SPN is valid, then we can perform exact inference in time linear to the graph's edges. Its syntax is different to other conventional models (read bayesian and markov networks) in the sense that its graph does not explicitly model events and (in) dependencies between variables. That is, whilst variables in a bayesian network are represented as nodes in the graph, with each edge connecting two nodes asserting a dependency relationship between the connected variables, a node in an SPN may not necessarily represent a variable or event, neither an edge connecting two nodes represent dependence. In this sense, SPNs can be seen as a type of probabilistic Artificial Neural Network (ANN). However, whilst neural networks represent a function, SPNs model a tractable probability distribution. Furthermore, SPNs are distinct from standard neural networks seeing that, whereas ANNs have only one type of neuron with an activation function mapping to values in $[0, 1]$, SPNs have two kind of neurons, which we will see in the next sections. Still, SPNs retain certain important characteristics from ANNs as we will discuss later, with mainly its deep structure properties [DB11] as the most interesting feature.

The Gens-Domingos Schema [GD13], or `LearnGD` as we will reference it throughout this paper, is an SPN structural learning algorithm proposed by Robert Gens and Pedro Domingos. Gens and Domingos call it a schema because it only provides a template of what the algorithm should be like. We will discuss `LearnGD` in details in the next section. This paper documents a particular implementation of the GD schema. Other implementations may have different results.

In this document, we show how we implemented the `LearnGD` algorithm. We analyze the complexity of each algorithm component in detail, later referring to such analyzes when drawing conclusions on the overall complexity of the algorithm. As we have mentioned before, since the `LearnGD` schema depends heavily on implementation, the complexity we achieve in this particular case may differ from other implementations. After each analysis, we then look at the algorithm as whole, drawing conclusions on time and memory usage, as well as implementation details that could potentially decrease the algorithm runtime. We also comment on how to implement better concurrency then how it is currently coded in our implementation. We then show some results on experiments made on image classification and image completion.

2. Sum-Product Networks

In this section we will define SPNs differently from other articles [GD13; PD11; DV12] as the original more convoluted definition is of little use for the `LearnGD` algorithm. Our definition is almost identical to the original `LearnGD` article [GD13], with the exception that we assume that an SPN is already normalized. This fact changes nothing, since Peharz *et al* recently proved that normalized SPNs have as much representability power as unnormalized SPNs [Peh+15]. Before we enunciate the formal definition of an SPN, we will give an informal, vague definition of an SPN in order to explain what completeness, consistency, validity and decomposability — which are an important set of definitions — of an SPN mean.

A sum-product network represents a tractable probability distribution through a DAG. Such digraph must always be weakly connected. A node can either be a leaf, a sum, or a product node. The scope of a node is the set of all variables present in all its descendants. Leaf nodes are tractable probability distributions and their scope is the scope of its distribution, sum nodes represent the summing out of the variables in its scope and product nodes act as feature hierarchy. An edge that has its origin from a sum node has a non-negative weight. We refer to a sub-SPN S rooted at node i as $S(i)$, while the SPN rooted at its root is denoted as $S(\cdot)$ or simply S. The scope of a node will be denoted as $\mathrm{Sc}(i)$, where i is a node. The set of children of a node will be denoted as $\mathrm{Ch}(i)$. Similarly, $\mathrm{Pa}(i)$ is the set of parents of node i.

Definition 2.1 (Normalized)**.**
Let S be an SPN and $\Sigma(S)$ be the set of all sum nodes of S. S is normalized iff, for all $\sigma \in \Sigma(S)$, $\sum_{c \in Ch(\sigma)} w_{\sigma c} = 1$ and $0 \leq w_{\sigma c} \leq 1$, where $w_{\sigma c}$ is the weight from edge $\sigma \to c$.

Definition 2.2 (Completeness).
Let S be an SPN and $\Sigma(S)$ be the set of all sum nodes of S. S is complete iff, for all $\sigma \in \Sigma(S)$, $Sc(i) = Sc(j), i \neq j; \forall i, j \in Ch(\sigma)$.

Definition 2.3 (Consistency).
Let S be an SPN, $\Pi(S)$ be the set of all product nodes of S and X a variable in $Sc(S)$. S is consistent iff X takes the same value for all elements in $\Pi(S)$ that contain X.

Definition 2.4 (Validity).
An SPN S is valid iff it always computes the correct probability of evidence S represents.

Theorem 2.1. *An SPN S is valid if it is both complete and consistent.*

Validity guarantees that the SPN will compute not only the correct probability of evidence, but also in time linear to its graph's edges. Therefore, it is preferable to learn valid SPNs. Notice that Theorem 2.1 is not restricted by completeness and consistency. In fact, incomplete and/or inconsistent SPNs can compute the probability of evidence correctly, but consistency and completeness guarantee that all sub-SPNs are also valid.

Definition 2.5 (Decomposability).
Let S be an SPN and $\Pi(S)$ be the set of all product nodes in S. S is decomposable iff, for all $\pi \in \Pi(S)$, $Sc(i) \cap Sc(j) = \emptyset, i \neq j; \forall i, j \in Ch(\pi)$.

It is clear that decomposability implies consistency, therefore if an SPN is both complete and decomposable, than it is also valid. We choose to work with decomposability because it is easier to learn decomposable SPNs then it is to learn consistent ones. We do not lose representation power because a complete and consistent SPN can be transformed into a complete and decomposable SPN in no more than a polynomial number of edge and node additions [Peh+15]. We can now formally define an SPN.

Definition 2.6 (Sum-product network).
A sum-product network (SPN) is a weakly connected DAG that can be recursively defined as following.

An SPN:

(1) *with a single node is a univariate tractable probability distribution (**leaf**);*
(2) *is a normalized weighted sum of SPNs of same scope (**sum**);*
(3) *is a product of SPNs with disjoint scopes (**product**).*

The value of an SPN is defined by its type. Let λ, σ and π be a leaf, sum and product respectively. The values of such SPNs are given by $\lambda(\mathbf{x})$, $\sigma(\mathbf{x})$ and $\pi(\mathbf{x})$, where $\mathbf{x}$ is a certain evidence instantiation.

Leaf: $\lambda(\mathbf{x})$ *is the value of the probability distribution at point* $\mathbf{x}$.
Product: $\pi(\mathbf{x}) = \prod_{c \in Ch(\pi)} c(\mathbf{x})$.
Sum: $\sigma(\mathbf{x}) = \sum_{c \in Ch(\sigma)} w_{\sigma c} c(\mathbf{x})$, *with* $\sum_{c \in Ch(\sigma)} w_{\sigma c} = 1$ *and* $0 \leq w_{\sigma c} \leq 1$.

Note that this definition assumes an SPN to be complete, decomposable and normalized. Other definitions in literature may differ from ours, but as we have mentioned before, for our implementation, this definition is convenient for us. Another observation worthy of notice is the value of $\lambda(\mathbf{x})$. Although here we consider $\mathbf{x}$ to be a multivariate instantiation (i.e. a set of — potentially multiple — variable valuations), we had initially defined a leaf to be a univariate distribution. Although it is possible to attribute leaves as multivariate probability distributions [RL14], for our definition we have chosen to keep a leaf's scope a unit set. Therefore, in the case of a leaf's value, $\mathbf{x}$ is a singleton (univariate) variable instantiation.

3. The `LearnGD` Schema

The `LearnGD` schema was proposed by Robert Gens and Pedro Domingos on *Learning the Structure of Sum-Product Networks* [GD13]. In this section we will outline the schema in pseudo-code and analyze a few properties derived from the algorithm.

Algorithm 1 `LearnGD`

Input Set $\mathbf{D}$ of instances (data)
Input Set $\mathbf{V}$ of variables (scope)
Output An SPN representing a probability distribution given by $\mathbf{D}$ and $\mathbf{V}$
1: **if** $|\mathbf{V}| = 1$ **then** ▷ univariate data sample
2:     **return** univariate distribution estimated from $T[\mathbf{V}]$ (data of $\mathbf{V}$)
3: **end if**
4: Take $\mathbf{V}$ and find mutually independent subsets $\mathbf{V}_i$ of variables
5: **if** possible to partition **then** ▷ i.e. we have found independent subsets
6:     **return** $\prod_i$ `LearnGD` $(\mathbf{D}, \mathbf{V}_i)$
7: **else** ▷ we cannot say there is independence
8:     Take $\mathbf{D}$ and find $\mathbf{D}_j$ subsets of similar instances
9:     **if** possible to partition **then**
10:         **return** $\sum_i \frac{|\mathbf{D}_j|}{|\mathbf{D}|} \cdot$ `LearnGD` $(\mathbf{D}_j, \mathbf{V})$
11:     **else** ▷ i.e. data is one big cluster
12:         **return** fully factorized distribution.
13:     **end if**
14: **end if**

Let us now, for a moment, suppose that SPNs are not necessarily complete, decomposable and normalized. We shall prove a few results derived from SPNs generated by Algorithm 1.

Lemma 3.1. *An SPN S generated by `LearnGD` is complete, decomposable and normalized.*

Proof. Lines 4–6 show that the scope of each child in a product node of S is a partition of the scope of their parent. Therefore, children have pairwise disjoint scopes on line 6, which proves decomposability for this part of the algorithm. In lines 8–10, since we are clustering similar instances, $\mathbf{D}$ is being partitioned but we are not changing $\mathbf{V}$ in any way. In fact, line 10 shows that we pass $\mathbf{V}$ to all other children. That is, all children of sum nodes have the same scope as their parent, which proves completeness. Let $\mathbf{D}_1, \ldots, \mathbf{D}_n$ be the subsets of similar instances. By the definition of clustering, $\mathbf{D}_1 \cup \ldots \cup \mathbf{D}_n = \mathbf{D}$ and $\mathbf{D}_i \cap \mathbf{D}_j = \emptyset$, $i \neq j$, $1 \leq i, j \leq n$. Thus it follows that $\sum_{i=1}^{n} \frac{|\mathbf{D}_i|}{|\mathbf{D}|} = 1$ and thus line 10 always creates complete and normalized sum nodes. Line 12 is a special case where, if we have discovered that $\mathbf{D}$ is one big data cluster, we shall create a product node π in which all children of π are leaves and

$$\bigcup_{\lambda \in \mathrm{Ch}(\pi)} \mathrm{Sc}(\lambda) = \mathrm{Sc}(\pi).$$

In other words, we fully factorize our product node into leaves. In this case, it is obvious that this product node is decomposable. □

`LearnGD` can be divided into four parts:

(1) Is the data univariate? If it is, return a leaf.
(2) Are partitions of the data independent? If they are, return a product node whose children are the independent partitions.
(3) Are partitions of the data similar? If they are, return a sum node whose children are the partition clusters.
(4) In case all else fails, we have a fully factorized distribution.

Going back to our definition of an SPN, we can now take a more intuitive approach and make the following observations:

(1) A leaf is nothing but a local/partitioned/sample distribution of a probability distribution given by a single variable.
(2) A product node determines independence between variables.
(3) A sum node is a clustering of similar data values (i.e. instances that are "alike").

This gives more semantic value to SPNs, whilst still retaining its expressivity. Following this approach, one can easily notice that each "layer" corresponds to a recursive call in `LearnGD`. In fact, each recursive call constructs a hidden layer that tries to partition the SPN even further. This gives SPNs a deep architecture that resembles deep models in that the deeper the model, the more representation power it has [DB11].

Let us now observe the scope of each type of node. A leaf is the trivial case, since it has a single variable in its scope by definition. Each layer above it can have either sum or product nodes. Let us now look at decomposability, that is: if a variable X appears in a child of a product node π, then X cannot appear in another child of π. This gives us the following result:

Lemma 3.2. *Let S be an SPN generated by `LearnGD`, and let $\Lambda(S)$ be the set of all leaves of S. Then, $\forall \lambda \in \Lambda(S)$, we have that, $\forall p \in Pa(\lambda)$, p is a product node.*

Proof. Our proof is by contradiction. Let us assume that $\exists p \in \mathrm{Pa}(\lambda)$ such that p is a sum node and $\exists c^* \in \mathrm{Ch}(p)$ a leaf. From our assumption that p is a sum node, we have that, since the SPN is complete, the scope of all children of p are the same and are all equal to the scope of p. Now let $c \in \mathrm{Ch}(p)$. There must exist another child c such that $c \neq c^*$ because of lines 5 and 9. From that we have $\mathrm{Sc}(c) = \mathrm{Sc}(c^*)$ because of completeness, and since $\mathrm{Sc}(c^*)$ is singular, then c must also be leaf. But it is impossible to have leaves with same scope and same parent (line 1 from Algorithm 1). Therefore, p is actually a product node. □

Lemma 3.3. *An SPN generated by `LearnGD` is a rooted tree.*

Proof. It suffices to show that for any vertex, its indegree is exactly one. We can prove that by saying that Algorithm 1 never adds edges between two already existing vertices. □

Definition 3.1. *A sum-product network that is a rooted tree is called a sum-product tree (SPT).*

Theorem 3.1.
Let S be an SPT generated by `LearnGD`. Let $n = |\,Sc(S)|$ and $m = |\mathbf{D}|$, where $\mathbf{D}$ is the data sent as parameter to `LearnGD`. Let h be the height of S. Then

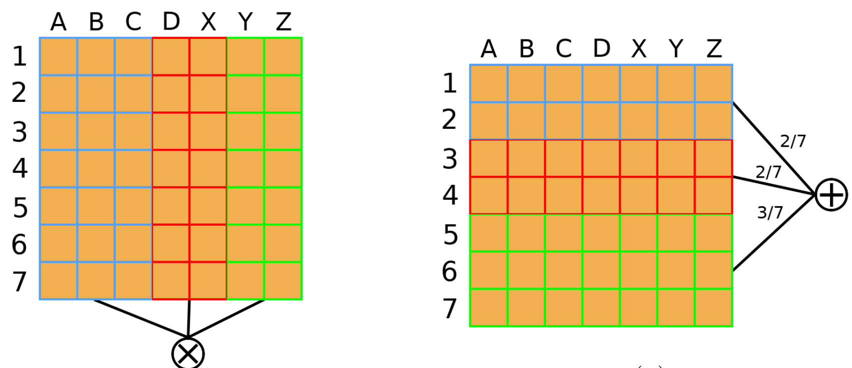
$$1 \leq h \leq n + m - 1$$

Proof. Sketch of proof: every sum or product node creates one more hidden layer (increments SPT's height by one) and decrements either an instance (if sum node) or variable in node's scope (if product node) by one at least. Last (deepest) sum node must have at least two product nodes as children (following Lemma 3.2), with each having one data instance each, therefore the number of layers created by sum nodes is at most $m - 1$. Similarly for product nodes, if we want to maximize the number of layers, the deepest product node must have two leaves as children, bringing the total count of product nodes to $n - 1$. Counting the last layer that is made out of leaves, we have $(n-1) + (m-1) + 1 = n + m - 1$. Furthermore, the SPT has the form of alternated sum-product layers (a sum node will follow a product node and vice-versa). This guarantees that each sum node modifies the overall data enough that the independence test will judge an independent variable from all others. The base case $h = 1$ is trivial: a size 1 scope generates one leaf with distribution equal to data. The case $h = 2$ is more interesting and occurs when all variables are dependent and belong to the same cluster.

TODO: Reword this more formally and unambiguously. Also put this after the variable independence test and data clustering sections. □

From Algorithm 1 we have learned that `LearnGD` can be structured into three parts. The first is discovering variable independencies and judging whether we should partition V and create a new product node. The second is, if the first part has failed, we must find possible clusters from the data we have. From these newly discovered clusters, we decide if we should create another sum node and assign each of its children a partition of these clusters, or if these clusters all form a single big all encompassing cluster. If this is the case, we create a new product node whose children are the fully factorized form of the present data. Finally, the third and last part is the base case. If the scope is of a single variable, we return the univariate probability distribution given by the univariate data.

If we were to visualize our dataset as a table where rows are instances and columns are variables, we could equate the algorithm as splitting, either horizontally or vertically, the dataset according to our partitioning decisions. For instance, if we had decided that there was a certain subset of variables that were independent of the rest of the variables, we would "split" our dataset table vertically, with each subtable belonging to a variable subset. Similarly for cluster partitioning, we would split our dataset table horizontally. Figure 1 illustrates the procedures for variable splitting (Figure 1a) and instance splitting (Figure 1b).



(A) (B)

Figure 1. These two images represent a dataset in table form. Rows are instances and columns are variables.Figure 1a shows variable splitting. In this example, we have observed that the subsets $\mathbf{V}_1 = \{A, B, C\}$, $\mathbf{V}_2 = \{D, X\}$ and $\mathbf{V}_3 = \{Y, Z\}$ are independent of each other. That is, for every pair $(P, Q), P \in \mathbf{V}_i, Q \in \mathbf{V}_j, i \neq j$, P is independent of variable Q. Given these partitions, we create a new product node whose children are the recursive calls to `LearnGD`. Note that their new scopes are now V_i, and their data instance covers only their new scope. In this example, the new partitions form subtables whose columns are adjacent to each other (e.g. A, B and C are adjacent). However, we could find an independent subset that did not necessarily obey a graphical rule, that is, we could have found that A and Y belong to the same partition. In this case, we would have considered A and Y as a new subset, regardless of their graphical positions. For Figure 1b, we apply a similar concept. In this case we are clustering similar instances. Note that instance partitioning does not alter their scope. Each instance subset $\mathbf{D}_1 = \{1, 2\}$, $\mathbf{D}_2 = \{3, 4\}$ and $\mathbf{D}_3 = \{5, 6, 7\}$ equates to a discovered cluster. A new sum node is then created, with weights corresponding to the ratio of rows in each subtable. The subtables are then added as children of the sum node and then recursed. Just like with variable splitting, these partitions do not necessarily obey a graphical rule. We could have non-adjacent rows as a single partition.

Now that we have the general idea of the algorithm, we shall describe and analyze how to do both variable and instance splitting. We will reserve a section to each of these topics. Once we have covered them both, we shall once again take a broader look at the `LearnGD` schema and work on some other results that depend on the two next sections.

4. Variable Independence

The core of variable splitting is finding independence between variables. What we wish to find is partitions of the current SPN scope such that every element in a partition is independent of all other elements in other partitions. In this section we shall explain the general idea, describe and analyze the method used in our implementation and lastly discuss certain problems encountered during experiments and implementation.

The description of our problem is: given a dataset with a set of variables $\mathbf{V}$, we wish to find a set $\mathbf{P} = \{\mathbf{P}_1, \mathbf{P}_2, \ldots, \mathbf{P}_n\}$, where $\mathbf{P}_i \cap \mathbf{P}_j = \emptyset, i \neq j$ and $\mathbf{P}_1 \cup \ldots \cup \mathbf{P}_n = \mathbf{V}$ and $\mathbf{P}_i$ is a subset of variables. That is, a set of partitions of $\mathbf{V}$. Additionally, for every $\mathbf{P}_i$ and $\mathbf{P}_j$, $i \neq j$, $\forall u \in \mathbf{P}_i, v \in \mathbf{P}_j$, $u \perp v$ (u is independent of v), where $\perp$ is the independence operator.

Suppose we have an independence oracle Ω that tells us if $X \perp Y$ and does so in constant time. A naïve solution to this problem is, for every pair $X, Y \in \mathbf{V}$, ask Ω if $X \perp Y$. We then memorize which ones are dependent and which are independent. For such memoization, we can use an undirected graph.

Definition 4.1 (Independence Graph)**.**
Let $G = (\mathbf{V}, \mathbf{E})$ be an undirected graph with vertex set $\mathbf{V}$ and edge set $\mathbf{E}$. Let i and j be vertices in $\mathbf{V}$. There exists an edge e_{ij} iff $i \not\perp j$.

This reduces our problem to one of finding connected subgraphs. Since there exists an edge if and only if the two connected variables are dependent, to find a partition $\mathbf{P}_i$, it suffices to find a connected subgraph in which all of its vertices have no path to another subgraph (there is no dependence path between variables).

Proposition 4.1. *If $H = (\mathbf{V}', \mathbf{E}')$ is a connected subgraph in Independence Graph $G = (\mathbf{V}, \mathbf{E})$ then $\mathbf{V}' \in \mathbf{P}$.*

Proof. Since there cannot be two same variables in our graph, it suffices to show that, $\forall u' \in \mathbf{V}', u \in \mathbf{V} \setminus \mathbf{V}'$: $u' \perp u$. Let us assume that $\mathbf{V}' \notin \mathbf{P}$. That is, there exists an element in $\mathbf{V}'$ that is dependent of an element in $\mathbf{V} \setminus \mathbf{V}'$. That means there exists a vertex $v' \in \mathbf{V}'$ and another vertex $v \in \mathbf{V} \setminus \mathbf{V}'$ such that an edge $e_{v'v}$ connects both of them. But $u \in \mathbf{V} \setminus \mathbf{V}'$ and $\mathbf{V}'$ is the vertex set of a connected subgraph, which means such an edge cannot exist, as there is no path from a vertex of H to a vertex in $\mathbf{V} \setminus \mathbf{V}'$. Therefore our assumption that $\mathbf{V}' \notin \mathbf{P}$ is false. $\square$

Sketching our naïve solution, we have:

Algorithm 2 `IndepGraph`

Input Set $\mathbf{D}$ of instances (data)
Input Set $\mathbf{V}$ of variables (scope)
Output A set $\mathbf{P}$ of independent partitions of variables

Let Ω be an independence oracle that returns true if independent and false if dependent
Let $G = (\mathbf{V}, \mathbf{E} = \emptyset)$ be the independence graph
for each variable $X \in \mathbf{V}$ **do**
    for each variable $Y \neq X, Y \in \mathbf{V}$ **do**
        if $\Omega(X, Y) =$ **false then**
            Let e_{XY} be a new edge connecting X and Y
            $\mathbf{E} \leftarrow \mathbf{E} \cup e_{XY}$
        end if
    end for
end for
$\mathbf{P} \leftarrow$`FindConnectedSubgraphs(G)`
return $\mathbf{P}$

We can find the connected subgraphs using a Union-Find structure. Since Union-Find is out of the scope of this paper, we shall not go into a deep discussion of it. However, we shall assume we have a Union-Find implementation which uses both *union by rank* and *path compression* heuristics, bringing the complexity of a series of m Union-Find operations to have amortized time $\mathcal{O}(m \log_2^*(n))$, where n is the number of elements and $\log^*$ is the iterated logarithm function.

Algorithm 3 `FindConnectedSubgraphs`

Input Graph $G = (\mathbf{V}, \mathbf{E})$, where $\mathbf{V}$ is vertex set and $\mathbf{E}$ is edge set
Output A set $\mathbf{S}$ of the sets of vertices of the connected subgraphs of G

Let $\mathbf{U}$ be the set of Union-Find structures.
for each variable $X \in \mathbf{V}$ **do**
    Let u be a new Union-Find structure whose representative is X.
    $u \leftarrow$ `MakeSet`(X)
    $\mathbf{U} \leftarrow \mathbf{U} \cup u$
end for
for each variable $X \in \mathbf{V}$ **do**
    for each variable $Y \in \mathbf{V}$ **do**
        $u_X \leftarrow$`Find`(X)
        $u_Y \leftarrow$`Find`(Y)
        if $u_X \neq u_Y$ **then** `Union`(u_X, u_Y)
        end if
    end for
end for
Let `Convert`$(\cdot)$ be function that converts Union-Finds to set of sets.
return `Convert`$(\mathbf{U})$

Let $n = |\mathbf{V}|$, the number of variables. The number of elements in all Union-Finds is n, as we can from lines 2–6. We also know from lines 7–14 that the number of Union-Find operations we call is $(2n)^2 + n(n-1) = 5n^2 - n$. Substituting these values into the amortized complexity of our Union-Find implementation gives

$$(5n^2 - n)\log_2^* n.$$

Since we are always differentiating edges $X - Y$ and $Y - X$, we can slightly improve performance by only taking into account one of those edges. We can decrease our number of Union-Find operations to $\binom{n}{2}$, which is the number of combinations if we choose 2 in n. The final complexity comes down to

$$\binom{n}{2}\log_2^* n = \left(\frac{n!}{(n-2)!2!}\right)\log_2^* n = \left(\frac{n(n-1)}{2}\right)\log_2^* n$$

Let $\alpha(n) = \log_2^* n$. We know that $\alpha(n)$ grows extremely slow, therefore we will assume $\alpha(n)$ as a constant, giving the final amortized complexity of Algorithm 3 the asymptotic form of $\mathcal{O}((n^2/2 - n/2)\log_2^* n) = \mathcal{O}(n^2\alpha(n) - n\alpha(n)) = \mathcal{O}(n^2)$.

Before we analyze `IndepGraph`, we need to review our previous assumption that a certain independence oracle Ω returns, in $\mathcal{O}(1)$, whether two variables are independent of each other given data. We will now show two independence tests we used for our implementation. The first is the standard Pearson's chi-square independence test and the second is the G-test. Both of these tests use the chi-square distribution. In our implementation we provide two options for computing the cumulative probability function (CDF) of a chi-square distribution of k degrees of freedom. One is based on the GNU Scientific Library (GSL) written in C, and the other is a Go implementation that calls Go's gamma function implementation. To simplify our analysis, we assume both of these implementations have same execution time $T(x)$. Furthermore, it is widely known that Pearson's and G-test's have same asymptotic time. In this paper we will assume all independence tests were run using the G-test. Therefore, when we refer to an "independence test" or "oracle", we mean we use a G-test to infer the dependency relation between two variables.

We shall call our independence test an `Oracle`. Our implementation is defined in pseudo-code as Algorithm 4. Let X be a variable and $\mathbf{D}$ be the dataset. We shall call $\text{Val}(X)$ the possible valuations of X. That is, if we had a variable *Animal* and we were only considering cats and dogs, we would then say that the possible valuations of *Animal* is defined by the set $\text{Val}(\textit{Animal}) = \{\textit{Cat}, \textit{Dog}\}$. Let $\mathbf{V}$ be the set of variables in our local scope, and $\mathbf{v}$ be the set of instantiations of $\mathbf{V}$. Let us denote the cartesian product $\underbrace{\mathbf{v} \times \cdots \times \mathbf{v}}_{n}$ as $\mathbf{v}^n$, and $N : \mathbf{v}^n \to \mathbb{Z}^*$, where N is a function that takes a set of valuations $\mathbf{v}$ and returns the number of instantiations that are consistent with $\mathbf{v}$ in $\mathbf{D}$. For instance, take our previous example where we have two possible valuations for *Animal*: *Cat* and *Dog*. Suppose our D shows we have three cats and four dogs in our data. Then $N[\textit{Animal} = \textit{Cat}] = 3$ and $N[\textit{Animal} = \textit{Dog}] = 4$. If we had a second variable *Behavior* with $\text{Val}(\textit{Behavior}) = \{\textit{Agitated}, \textit{Calm}\}$, and D showed:

Animal	**Behavior**
Dog	Agitated
Cat	Calm
Cat	Agitated
Dog	Agitated
Cat	Calm
Dog	Calm
Dog	Agitated

We could say that $N[Animal = Cat, Behavior = Calm] = 2/3$. Algorithm 4 builds a contingency table from data D, computing the N counts of each possible instantiations of the two populations. We call these counts the observed frequencies. We then compute the expected frequencies from the ratios of the totals. Next we compute $G = 2\sum_i O_i \cdot \ln\left(\frac{O_i}{E_i}\right)$, find the CDF of G in the corresponding chi-square distribution and compare with a certain significance value. If the area found on the chi-square distribution is less than the significance value, than we can, under the null hypothesis, reject the fact that they are independent.

Let A be an $m \times n$ matrix with elements denoted by a_{ij}. We shall use the notation $A[i:j][p:q]$ as the submatrix derived from taking the columns $[a_{ip}, \ldots, a_{iq}]$, …, $[a_{jp} \ldots, a_{jq}]$ from A. To simplify our algorithm, we assume $\mathrm{Val}(X)$ is an ordered set and is indexed from 1.

Algorithm 4 `Oracle`

Input Variables X and Y
Input Dataset D
Output Returns the result of the evaluation $X \perp Y$

Let $p = |\,\mathrm{Val}(X)|$ and $q = |\,\mathrm{Val}(Y)|$
Let C be a $(p+1) \times (q+1)$ matrix
Let C^* be the submatrix $C[1:p][1:q]$
for $i \leftarrow 1$ to p **do**
    $\nu_X \leftarrow \mathrm{Val}(X)[i]$
    for $j \leftarrow 1$ to q **do**
        $\nu_Y \leftarrow \mathrm{Val}(Y)[j]$
        $c_{ij} \leftarrow N[X = \nu_X, Y = \nu_Y]$
        $c_{(p+1,j)} \leftarrow N[Y = \nu_Y]$
    end for
    $c_{(i,q+1)} \leftarrow N[X = \nu_X]$
end for
$c_{(p+1,q+1)} \leftarrow N[X = \cdot, Y = \cdot]$ ▷ Total number of instances
$E_{ij} = \frac{c_{(p+1,j)}c_{(i,q+1)}}{c_{(p+1,q+1)}}$, for $i = 1, \ldots, p$ and $j = 1, \ldots, q$
$g \leftarrow \sum_{i=1}^{p}\sum_{j=1}^{q} c_{ij} * \ln\left(\frac{c_{ij}}{E_{ij}}\right)$
Let F be the CDF for $\chi^2((p-1)\cdot(q-1))$, and σ be the significance value
return $F(g) \geq \sigma$

Algorithm 4 has time $\mathcal{O}(|Val(X)| \cdot |Val(Y)| + T(m))$. If we consider a series of `Oracle` operations on all variables, we can conclude that our worst time asymptotically would be $\mathcal{O}(m^2 + T(m))$, where $m = \max |Val(V)|$. Going back to Algorithm 2, we now have that our independence oracle no longer takes time $\mathcal{O}(1)$. In fact, our `IndepGraph` routine has time

$$n^2 \cdot (m^2 T(m)) + \frac{n(n-1)}{2} \log_2^* n$$

where $n = |\mathbf{V}|$. Let us assume, for simplicity, that $\mathcal{O}(T(x)) = \mathcal{O}(1)$. The asymptotic time is then

$$\mathcal{O}(n^2 m^2 + \frac{n(n-1)}{2} \log_2^* n) = \mathcal{O}(n^2 m^2 + n^2) = \mathcal{O}(n^2 m^2)$$

References

[DB11] Olivier Delalleau and Yoshua Bengio. "Shallow vs. Deep Sum-Product Networks". In: *Advances in Neural Information Processing Systems 24 (NIPS 1011)* (2011).

[DV12] Aaron Dennis and Dan Ventura. "Learning the Architecture of Sum-Product Networks Using Clustering on Variables". In: *Advances in Neural Information Processing Systems* 25 (2012).

[GD13] Robert Gens and Pedro Domingos. "Learning the Structure of Sum-Product Networks". In: *International Conference on Machine Learning* 30 (2013).

[PD11] Hoifung Poon and Pedro Domingos. "Sum-Product Networks: A New Deep Architecture". In: *Uncertainty in Artificial Intelligence* 27 (2011).

[Peh+15] Robert Peharz et al. "On Theoretical Properties of Sum-Product Networks". In: *International Conference on Artificial Intelligence and Statistics 18 (AISTATS 2015)* (2015).

[RL14] Amirmohammad Rooshenas and Daniel Lowd. "Learning Sum-Product Networks with Direct and Indirect Variable Interactions". In: *International Conference on Machine Learning 31 (ICML 2014)* (2014).